Expressão Regular

regex.jpg

Expressão Regular

Histórico

A origem das expressões regulares estão na teoria dos autômatos e na teoria das linguagens formais, e ambas fazem parte da teoria da computação. Esses campos estudam modelos de computação (autômatas) e formas de descrição e classificação de linguagens formais. Na década de 1950, o matemático Stephen Cole Kleene descreveu tais modelos usando sua notação matemática chamada de "conjuntos regulares", formando a álgebra de Kleene. A linguagem SNOBOL foi uma implementação pioneira de casamento de padrões, mas não era idêntica às expressões regulares. Ken Thompson construiu a notação de Kleene no editor de texto QED como uma forma de casamento de padrões em arquivos de texto. Posteriormente, ele adicionou essa funcionalidade no editor de texto Unix ed, que resultou no uso de expressões regulares na popular ferramenta de busca grep. Desde então, diversas variações da adaptação original de Thompson foram usadas em Unix e derivados, incluindo expr, AWK, Emacs, vi e lex.

As expressões regulares de Perl e Tcl foram derivadas da biblioteca escrita por Henry Spencer, e no Perl a funcionalidade foi expandida posteriormente.[1] Philip Hazel desenvolveu a PCRE (Perl Compatible Regular Expressions), uma biblioteca usada por diversas ferramentas modernas como PHP e o servidor Apache. Parte do desenvolvimento do Perl 6 foi melhorar a integração das expressões regulares de Perl, e aumentar seu escopo e funcionalidade para permitir a definição de gramáticas de expressão de analisadores sintáticos.[2] O resultado foi uma mini-linguagem, as regras do Perl 6, usada para definir a gramática do Perl 6 assim como fornecer uma ferramenta para programadores da linguagem. Tais regras mantiveram as funcionalidades de expressões regulares do Perl 5.x, mas também permitiram uma definição BNF de um analisador sintático descendente recursivo.

O uso de expressões regulares em normas de informação estruturada para a modelagem de documentos e bancos de dados começou na década de 1960, e expandiu na década de 1980 quando normas como a ISO SGML foram consolidadas.

Aplicação em Ruby

Exemplo 1

Começaremos com a seguinte regex:

```
(19|20)\d\d([- /.])(0[1-9]|1[012])\2([012][0-9]|3[01])
```

E você se pergunta: "O que eu faço com isso???"

Calma, calma. Vamos destrinchar cada parte dela e entender pra quê serviria.

Na regex temos:

1. (19 | 20) é um grupo que casa "OU dezenove OU vinte";
2. \d representa um dígito "0 a 9";
3. ([- /.]) é um grupo que casa "hífen", "espaço", "barra" ou "ponto";
4. (0 [1-9] | 1 [012]) é um grupo quer dizer que ele casará "zero" seguido de "um até nove" OU "um" seguido de "zero", "um" ou "dois";
5. \2 significa que ele re-utiliza o segundo grupo da regex ([- /.]);
6. ([012] [0-9] | 3 [01]) é um grupo que casa "zero", "um" ou "dois" seguido de um dos dígitos OU "três" seguido de "zero" ou "um";

Ou seja, temos uma regex que casaria as datas dentro deste texto:

```
1954-10-01 João Alberto
1976-07-25 Maria Eduarda
1966-10-22 Carlos Silva
```

Poderíamos substituir partes dessa regex para sermos mais específicos, como os itens 1. e 2. por um ano, 4. por um mês e 6. por um dia. Falta validação pra evitar, por exemplo, que os dias 02-[30|31] sejam encontrados.

Exemplo 2

Vejamos essa outra regex

```
\A([^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\Z
```

Temos nela:

1. \A que indica o começo da cadeia de caracteres;
2. ([^@\s] +) é um gupo que casa qualquer caractere menos "arroba" e "espaço", ocorrendo 1 ou + vezes;
3. @ casa "arroba";
4. ?: quer dizer que esse grupo não pode ser re-utilizado;
5. [- a-z 0-9]+ \. diz que podem conter "traço", alfabeto em caixa baixa e dígito uma ou mais vezes seguido de "ponto";
6. + (guloso) diz que esta sequência deve acontecer uma ou mais vezes;
7. [a-z] {2, } diz que deve conter 2 ou mais letras do alfabeto em caixa baixa;
8. \Z indica o fim da cadeia de caracteres;

Essa regex seria uma das possíveis para localizar um endereço de email dentro de um texto. Por exemplo:

```
Perfil

Nome: José da Silva
Endereço: Rua 1, Quadra 2, Lote 3, Casa 4
E-mail: josesilva@mail.com.br
Website: www.empresa1.com.br
```

Exemplo 3

Vejamos mais uma com aplicação direta em ruby:

```
reg = Regexp.new("/^(?=.*\d)(?=.*([a-z]|[A-Z]))([\x20-\x7E]){8
ou
reg = %r(^(?=.*\d)(?=.*([a-z]|[A-Z]))([\x20-\x7E]){8,40}$)
ou ainda
reg = /^(?=.*\d)(?=.*([a-z]|[A-Z]))([\x20-\x7E]){8,40}$/
```

Sendo

1. / ... / para delimitar a regex
2. ^ ... $ para dizer que essa cadeia de caracteres ocupa uma linha inteira. ^ --> início / $ --> fim
3. ?= diz que esse grupo servirá pra validar, mas não casará nem será re-utilizado;
4. .*\d diz que haverá um dígito, sozinho ou com algo antes;
5. (?= .* ([a-z] | [A-Z])) tb só valida e identifica se existe alguma letra na cadeia;
6. ([\x20 - \7E]) diz que a cadeia pode ter os caracteres especiais ASCII x20 até x7E;
7. { 8,40 } delimita a quantidade de caracteres da cadeia, entre 8 e 40;

Hummm... isso tá com cara de ser um validador de senha ... :D . Segundo essa regex a senha TEM que ter números e letras e entre 8 e 40 caracteres. A função no Ruby ficaria assim:

```
def validate_password(password)

  reg = /^(?=.*\d)(?=.*([a-z]|[A-Z]))([\x20-\x7E]){8,40}$/

return (reg.match(password))? true : false

end
```

No Rails ficaria assim:

```
class MyModel

  validates_format_of :password, :with => /^(?=.*\d)(?=.*([a-z]|[A-Z]))(|

end
```

Como podemos ver, existem várias maneiras de se casar caracteres, umas mais específicas e outras mais gerais. Para alguns casos algumas não servem. É necessário, porém, verificar se a expressão casa ou deixa de casar alguns caracteres específicos.

"Glossário"

Tabela de metacaracteres

O que é	O que faz	Exemplo (Busca)	Resultado
. (ponto)	Curinga (todos caracteres)	.ato	aato, Bato, -ato, ?ato, _ato, ...
[ ] (abre-fecha colchetes)	Lista (só os chars dentro dele)	[fgpr]ato	fato, gato, pato, rato
- (traço)	Escopo (De - até)	[A-Z]	A, B, C, D, E, F, G, H, ...
^ (circumflexo)	Negação (não-alguma-coisa)	[^fg]ato	(sem "f" nem "g") pato, rato
	Início da linha (começa com ...)	^[CFT]aça	"Caça ...", "Faça ...", "Taça ..."
? (interrogação)	Opcional (0 ou 1 alguma-coisa)	pr?ato	pato, prato
***** (asterisco)	Qualquer (0, 1 ou infinitas vezes)	go*l	gl, gol, gool, goooooooool
+ (mais)	Existente (1 ou mais vezes)	cor+ta	corta, corrta, corrrrrrrrrrta
{ } (abre-fecha chaves)	Escopo de repetições	ol{2,5}á	ollá, olllá, ollllá, olllllá
$ (cifrão)	Final da linha (termina com ...)	hoje$	"... fui hoje", "... só hoje"
\b (barra invertida + B)	Borda da palavra	\bclara\b	palavra exata "clara" (not claras)
**** (barra invertida)	Escape (para chars especiais)	\. , \[, \] , \^ , \? , * , \+ , ...	os metacaracteres especiais
| (pipeline)	Ou	foca | toca	foca ou toca
( ) (abre-fecha parêntesis)	Grupo (conjunto de caracteres)	((su|hi)per)?cão	cão, supercão, hipercão
\[1-9] (barra invertida + 1 até 9)	Retrovisor (reusa um grupo)	(tico)-\1 | (teco)-\2\1	tico-tico ou teco-tecotico

Tabela de classes de caracteres

Qual?	O que casa?	Generalizando	Exemplos
[:alnum:]	Caracteres alfanuméricos	[0-9a-zA-Z]	Conheci; 800mil; opAAaa

Qual?	O que casa?	Generalizando	Exemplos
[:alpha:]	Caracteres alfabéticos	[A-Za-z]	RorRocks
[:blank:]	Espaço e tabulação	[\t]	" "
[:cntrl:]	Caracteres de controle	[\x00-\x1F\x7F]	Não me pergunte ...
[:digit:] ou \d	Dígitos	[0-9]	4; 8; 15; 16 ;23 ;42 ...
[:graph:]	Caracteres visíveis	[\x21-\x7E]	Tb não tenho nem idéia
[:lower:]	Caracteres em caixa baixa	[a-z]	abcdef ...
[:print:]	Caracteres visíveis e espaços	[\x20-\x7E]	começando a fazer sentido
[:punct:]	Caracteres de pontuação	[^\w\s]	:); =(;
[:space:] ou \s	Caracteres de espaços em branco	[\t \r \n \v \f]	"espaço"; "return"; "quebra";
[:upper:]	Caracteres em caixa alta	[A-Z]	ABCDEF...
[:xdigit:]	Dígitos hexadecimais	[A-Fa-f0-9]	FaCaAf14dA; 0033CC;
\S	Tudo menos espaços em branco	[^[:space:]]	aZ1.,']);!@$/
\w	Letras, dígitos e ' '	[a-zA-Z0-9]	aB10 as;
\W	Contrário de \w	[^\w]	@./;*&

Referências

Wikipedia pt-BR

Wikipedia en

Guia do Sourceforge para ER

Guia de RegEx pro VIM

Testador de Regex do Ruby Online

Regex para várias linguagens e editores

Exemplo 3